10 网络编程

10.1 Berkeley Socket

TCP/IP协议族标准只规定了网络各个层次的设计和规范，具体实现则需要由各个操作系统厂商完成。最出名的网络库由BSD 4.2版本最先推出，所以称作**伯克利套接字**，这些API随后被移植到各大操作系统中，并成为了网络编程的事实标准。`socket`即套接字是指网络中一种用来建立连接、网络通信的设备，用户创建了`socket`之后，可以通过其发起或者接受`TCP`连接、可以向`TCP`的发送和接收缓冲区当中读写TCP数据段，或者发送`UDP`文本。

10.2 地址信息设置

struct sockaddr 和 struct sockaddr_in

我们主要以IPv4为例介绍网络的地址结构。主要涉及的结构体有`struct in_addr`、`struct sockaddr`、`struct sockaddr_in`。其中`struct sockaddr`是一种通用的地址结构，它可以描述一个IPv4或者IPv6的结构，所有涉及到地址的接口都使用了该类型的参数，但是过于通用的结果是直接用它来描述一个具体的IP地址和端口号十分困难。所以用户一般先使用`struct sockaddr_in`来构造地址，再将其进行强制类型转换成`struct sockaddr`以作为网络接口的参数。

```
//man 7 ip
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

大小端转换

TCP/IP协议规定，当数据在网络中传输的时候，一律使用**网络字节序**即**大端法**。对于应用层协议的载荷部分，如果不需要第三方工具检测内容，可以不进行大小端转换（因为接收方和发送方都是**主机字节序**即**小端法**）。但是对于其他层次的头部部分，在发送之前就一定要进行小端到大端的转换了（因为网络中的通信以及`tcpdump`、`netstat`等命令都是以大端法来解析内容的）。

下面是整数大小端转换相关的函数。

```
#include <arpa/inet.h>
//h --> host n --> net l --> 32bit s --> 16bit
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

下面是示例代码：

```
int main(){
    unsigned short us = 0x1234;
    printf("%x\n",us);
    unsigned short us1 = htons(us);//将端口号从小端转换成大端
    printf("%x\n",us1);
    printf("%x\n",ntohs(us1));
    return 0;
}
```

下面是32位网络字节序IP地址和点分十进制的IP地址互相转换的函数：

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
//线程安全版本是inet_atop inet_ptoa
```

下面是示例代码：

```c
int main(int argc,char* argv[])
{
    //./inet_aton 127.0.0.1
    struct sockaddr_in addr;
    inet_aton(argv[1],&addr.sin_addr);//将点分十进制转换成32位网络字节序
    printf("addr = %x\n",addr.sin_addr.s_addr);
    printf("addr = %s\n",inet_ntoa(addr.sin_addr));//将32位网络字节序
转换成点分十进制
    return 0;
}
```

域名和IP地址的对应关系

IP层通过IP地址的结构进行路由选择最终找到一条通往目的地的路由，但是一些著名的网站如果采用IP地址的方式提供地址，用户将无法记忆，所以更多的时候需要一个方便人类记忆的**域名**（比如www.kernel.org）作为其实际IP地址（145.40.73.55）的别名，显然我们需要一种机制去建立域名和IP地址的映射关系，一种方法是修改本机的hosts文件`/etc/hosts`，但是更加通用的方案是利用DNS协议，去访问一个DNS服务器，服务器当中存储了域名和IP地址的映射关系。与这个操作相关的函数是`gethostbyname`，下面是其用法：

```c
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
                char  *h_name;            /* official name of host
*/
                char **h_aliases;         /* alias list */
                int    h_addrtype;        /* host address type */
                int    h_length;          /* length of address */
                char **h_addr_list;       /* list of addresses */
            }
```

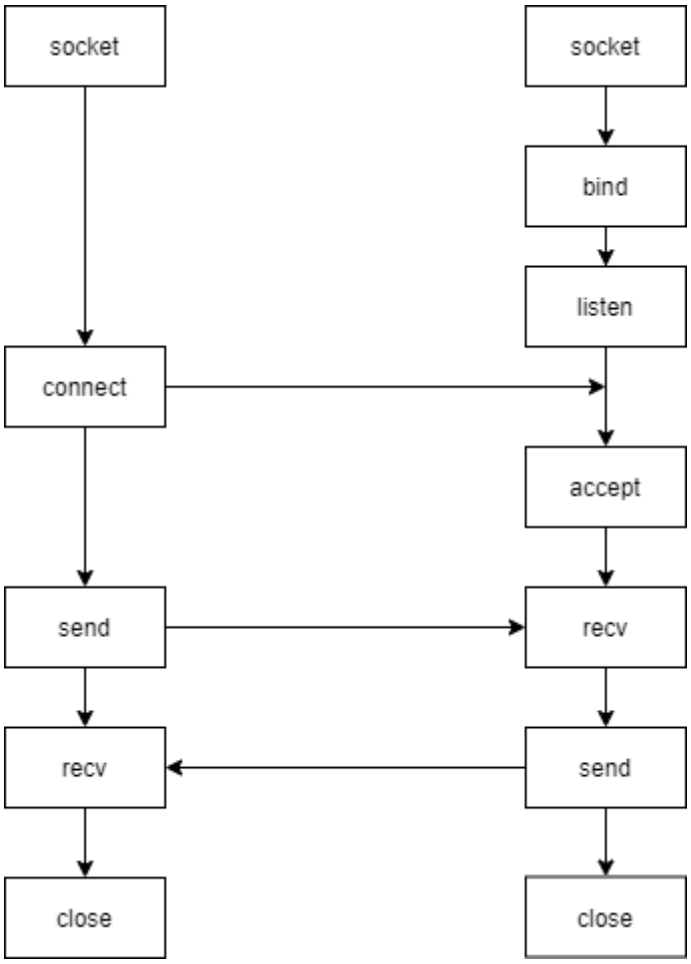
```c
int main(int argc, char *argv[]){
    ARGS_CHECK(argc,2);
    struct hostent* pHost = gethostbyname(argv[1]);
    ERROR_CHECK(pHost,NULL,"gethostbyname");
    printf("hname=%s\n",pHost->h_name);//真实主机名
    for(int i=0;pHost->h_aliases[i] != NULL;i++){
```

```
        printf("aliases=%s\n",pHost->h_aliases[i]);//别名列表
    }
    printf("addrtype=%d\n",pHost->h_addrtype);//地址类型
    printf("addrlength=%d\n",pHost->h_length);//地址长度
    char buf[128]={0};
    for(int i=0;pHost->h_addr_list[i] != NULL;i++){
        memset(buf,0,sizeof(buf));
        inet_ntop(pHost->h_addrtype,pHost-
>h_addr_list[i],buf,sizeof(buf));
        printf("addr=%s\n",buf);
    }
    return 0;
}
/*
$ ./gethostbyname www.baidu.com
hname=www.a.shifen.com
aliases=www.baidu.com
addrtype=2
addrlength=4
addr=14.215.177.38
addr=14.215.177.39
*/
```

10.3 TCP通信

鸟瞰TCP通信

下面是使用TCP通信的流程图:



socket

`socket`函数用于创建一个`socket`设备。调用该函数时需要指定通信的协议域、套接字类型和协议类型。一般根据选择TCP或者UDP有着固定的写法。`socket`函数的返回值是一个非负整数，就是指向内核`socket`设备的文件描述符。

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
//domain AF_INET --> IPv4 AF_INET6 --> IPv6
//type SOCK_STREAM --> TCP SOCK_DGRAM --> UDP
//protocol IPPROTO_TCP --> TCP IPPROTO_UDP -->UDP
```

每个连接的一端在内核中都对应了一个输入缓冲区(SO_RCVBUF)和一个输出缓冲区(SO_SNDBUF)，其默认大小由`/proc/sys/net/core/rmem_default`和`/proc/sys/net/core/wmem_default`文件指定。

connect

客户端使用`connect`来建立和TCP服务端的连接。

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t
addrlen);
```

客户端在调用`connect`可以不使用`bind`来指定本地的端口信息，这客户端就会随机选择一个临时端口号来作为源端口。调用`connect`预期是完成TCP建立连接的三次握手。如果服务端未开启对应的端口号或者未监听，则只能收到一个RST回复，并且报错返回的内容是"Connection refused"。

```
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = connect(sfd,(struct
sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"connect");
    return 0;
}
```

使用`tcpdump`命令可以查看包的状态。

```
-- 发起connect
$ ./client_once 127.0.0.1 2778
connect: Connection refused
-- 使用root抓包，注意观察Flags
#tcpdump -i any port 2778
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size
262144 bytes
22:51:12.289046 IP localhost.33004 > localhost.2778: Flags [S], seq
4122161411, win 65495, options [mss 65495,sackOK,TS val 2046136533
ecr 0,nop,wscale 7], length 0
22:51:12.289440 IP localhost.2778 > localhost.33004: Flags [R.],
seq 0, ack 4122161412, win 0, length 0
```

bind

`bind`函数用于给套接字赋予一个**本地**协议地址（即IP地址加端口号）。`bind`在选择端口号时应当避开知名端口号的范围(<1024)。

```
int bind(int sockfd, const struct sockaddr *addr,
                 socklen_t addrlen);
```

使用`bind`函数时要注意其地址是大端法描述的，并且可能需要执行强制类型转换。

```
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    //创建监听套接字
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);//如果赋值为
INADDR_ANY，表示选择本机IP地址
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
```

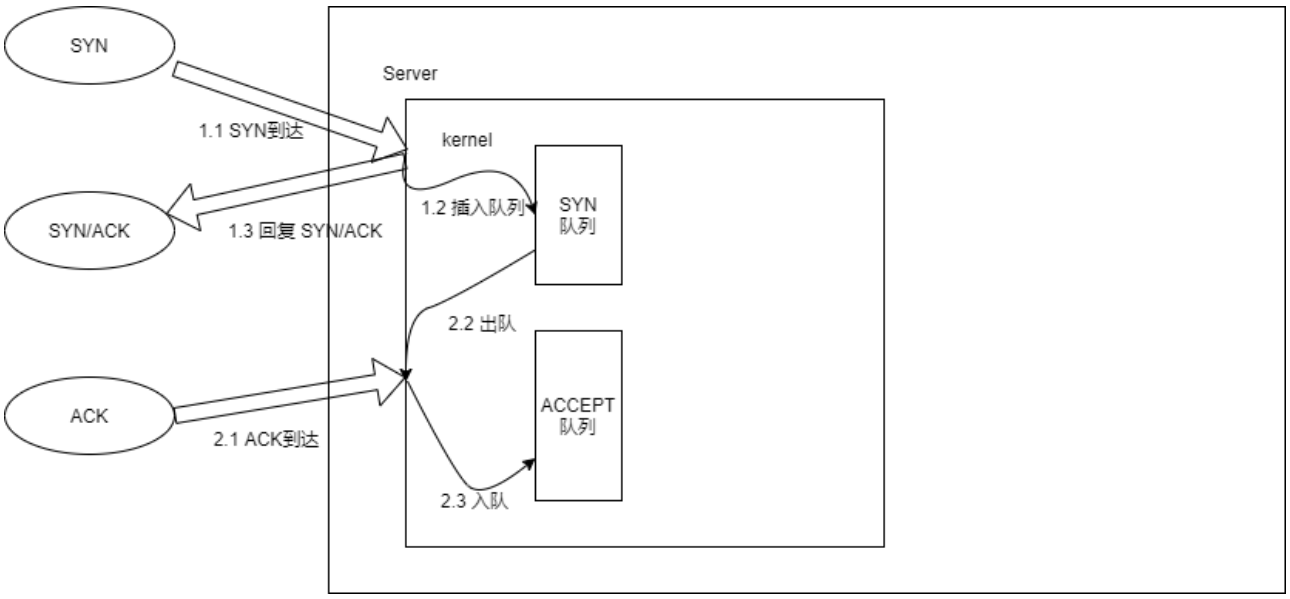
```
    ERROR_CHECK(ret,-1,"bind");
    return 0;
}
```

listen

使TCP服务端开启监听。服务端在开启了`listen`之后，就可以开始接受客户端连接了。

```
int listen(int sockfd, int backlog);
```

一旦启用了`listen`之后，操作系统就知道该套接字是服务端的套接字，操作系统内核就不再启用其发送和接收缓冲区，转而在内核区维护两个队列结构：**半连接队列**和**全连接队列**。半连接队列用于管理成功第一次握手的连接，全连接队列用于管理已经完成三次握手的队列。`backlog`在有些操作系统用来指明半连接队列和全连接队列的长度之和，一般填一个正数即可。如果队列已经满了，那么服务端受到任何再发起的连接都会直接丢弃（大部分操作系统中服务端不会回复RST，以方便客户端自动重传）



使用`netstat -an`命令可以查看主机上某个端口的监听情况。

```
-- 下面是listen之后的内容
$ netstat -an|grep 2778
tcp        0      0 127.0.0.1:2778          0.0.0.0:*
LISTEN
```

DDOS攻击

利用半连接队列的设计思路，网络攻击者想到了一种恶意攻击的方法。他们伪造一些SYN请求但是并不打算建立连接，这些请求的源地址随机构建的，或者是感染其他计算机（即“肉鸡”）来发起请求，服务端内核就会维持一个很大的队列来管理这些半连接。当半连接足够多的时候，就会导致新来的正常连接请求得不到响应， 也就是所谓的**DDOS攻击**。

一般的防御措施就是就是减小SYN+ACK重传次数、增加半连接队列长度、启用syn cookie。不过在高强度攻击面前，调整tcp_syn_retries 和 tcp_max_syn_backlog并不能解决根本问题。更有效的防御手段是激活tcp_syncookies——在连接真正创建起来之前，它并不会立刻给请求分配数据区存储连接状态，而是通过构建一个带签名的序号来屏蔽伪造请求。

accept

`accept`函数由服务端调用，用于从全连接队列中取出下一个已经完成的TCP连接。如果全连接队列为空，那么`accept`会陷入阻塞。一旦全连接队列中到来新的连接，此时`accept`操作就会就绪，这种就绪是**读操作就绪**，所以可以使用`select`函数的读集合进行监听。当`accept`执行完了之后，内核会创建一个**新的套接字文件对象**，该文件对象关联的文件描述符是`accept`的返回值，文件对象当中最重要的结构是一个发送缓冲区和接收缓冲区，可以用于服务端通过TCP连接发送和接收TCP段。

区分两个套接字是非常重要的。通过把旧的管理连接队列的套接字称作**监听套接字**，而新的用于发送和接收TCP段的套接字称作**已连接套接字**。通常来说，监听套接字会一直存在，负责建立各个不同的TCP连接（只要源IP、源端口、目的IP、目的端口四元组任意一个字段有区别，就是一个新的TCP连接），而某一条单独的TCP连接则是由其对应的已连接套接字进行数据通信的。

客户端使用`close`关闭套接字或者服务端使用`close`关闭已连接套接字的时候就是主动发起断开连接四次挥手的过程。

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

需要特别注意的是，`addrlen`参数是一个传入传出参数，所以使用的时候需要主调函数提前分配好内存空间。

```
// ...
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    //newFd代表的是跟客户端的TCP连接
    socklen_t len;
    len = sizeof(struct sockaddr)
    int newFd = accept(sfd,NULL, &len);
    ERROR_CHECK(newFd,-1,"accept");
    printf("newFd = %d\n", newFd);
// ...
```

send和recv

`send`和`recv`用于将数据在用户态空间和内核态的缓冲区之间进行传输，无论是客户端还是服务端均可使用，但是只能用于TCP连接。将数据拷贝到内核态并不意味着会马上传输，而是会根据时机再由内核协议栈按照协议的规范进行分节，通常缓冲区如果数据过多会分节成MSS的大小，然后根据窗口条件传输到网络层之中。

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

使用`read`和`write`可以实现同样的效果，相当于`flags`参数为0。

下面是一个完整的客户端和服务端通信的例子：

```
//clinet.c
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = connect(sfd,(struct
sockaddr*)&serAddr,sizeof(serAddr));
```

```
    ERROR_CHECK(ret,-1,"connect");
    char buf[64]={0};
    ret = recv(sfd,buf,sizeof(buf),0);
    printf("buf=%s\n",buf);
    send(sfd,"helloserver",11,0);
    close(sfd);
    return 0;
}
```

```
//server.c
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    //创建监听套接字
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    //newFd代表的是跟客户端的TCP连接
    socklen_t len;
    len = sizeof(struct sockaddr);
    int newFd = accept(sfd,NULL,&len);
    ERROR_CHECK(newFd,-1,"accept");
    ret = send(newFd,"helloclient",11,0);
    ERROR_CHECK(ret,-1,"send");
    char buf[64]={0};
    ret = recv(newFd,buf,sizeof(buf),0);
    printf("buf=%s\n",buf);
    close(newFd);
    close(sfd);
    return 0;
}
```

需要特别注意的是，`send`和`recv`的次数和网络上传输的TCP段的数量没有关系，多次的`send`和`recv`可能只需要一次TCP段的传输。另外一方面，TCP是一种流式的通信协议，消息是以字节流的方式在信道中传输，这就意味着一个重要的事情，**消息和消息之间是没有边界的**。在不加额外约定的情况下，通信双方并不知道发送和接收到底有没有接收完一个消息，有可能多个消息会在一次传输中被发送和接收（江湖俗称"粘包"），也有有可能一个消息需要多个传输才能被完整的发送和接收(江湖俗称"半包")。

```
//...
    /* char buf[64]={0}; */
    /* ret = recv(newFd,buf,sizeof(buf),0); */
    char buf[4]={0};//修改recv读取的单位，可能会产生"半包"问题
    ret = recv(newFd,buf,sizeof(buf),0);
    printf("ret=%d,buf=%s\n",ret,buf);
    ret = recv(newFd,buf,sizeof(buf),0);
    printf("ret=%d,buf=%s\n",ret,buf);
//...
```

使用select实现TCP即时聊天

基于TCP的聊天程序的实现思路和之前利用管道实现即时聊天的思路是一致的。客户端和服务端都需要使用`select`这种IO多路复用机制监听读事件，客户端需要监听套接字的读缓冲区以及标准输入，服务端需要监听已连接套接字的读缓冲区以及标准输入。

```
//server.c
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    int newFd = accept(sfd,NULL,NULL);
    ERROR_CHECK(newFd,-1,"accept");
```

```
    char buf[64]={0};
    fd_set rdset;
    while(1){
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(newFd,&rdset);
        select(newFd+1,&rdset,NULL,NULL,NULL);
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            memset(buf,0,sizeof(buf));
            int ret = read(STDIN_FILENO,buf,sizeof(buf));
            if(ret == 0){
                break;
            }
            send(newFd,buf,strlen(buf)-1,0);
        }
        else if(FD_ISSET(newFd,&rdset)){
            memset(buf,0,sizeof(buf));
            ret = recv(newFd,buf,sizeof(buf),0);
            //对端断开的时候，newFd一直可读
            //recv读数据的返回值是0
            if(0 == ret){
                printf("byebye\n");
                close(sfd);
                close(newFd);
                return 0;
            }
            printf("buf=%s\n",buf);
        }
    }
    close(newFd);
    close(sfd);
    return 0;
}
```

```
//client.c
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
```

```c
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = connect(sfd,(struct
sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"connect");
    char buf[64]={0};
    fd_set rdset;
    while(1){
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(sfd,&rdset);
        select(sfd+1,&rdset,NULL,NULL,NULL);
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            memset(buf,0,sizeof(buf));
            ret = read(STDIN_FILENO,buf,sizeof(buf));
            if(ret == 0){
                break;
            }
            send(sfd,buf,strlen(buf)-1,0);
        }
        else if(FD_ISSET(sfd,&rdset)){
            memset(buf,0,sizeof(buf));
            ret = recv(sfd,buf,sizeof(buf),0);
            if(ret == 0){
                break;
            }
            printf("buf=%s\n",buf);
        }
    }
    close(sfd);
    return 0;
}
```

TIME_WAIT和setsockopt

如果是服务端主动调用`close`断开的连接，即服务端是四次挥手的主动关闭方，由之前的TCP状态转换图可知，主动关闭方在最后会处于一个固定2MSL时长的TIME_WAIT等待时间。在此状态期间，如果尝试使用`bind`系统调用对重复的地址进行绑定操作，那么会报错。

```
$ ./server_tcpchat1 192.168.135.132 2778
bind: Address already in use
$ netstat -an|grep 2778
tcp        0      0 192.168.135.132:2778    192.168.135.133:57466
TIME_WAIT
```

在实际工作当中，TIME_WAIT状态的存在虽然有可能会提高连接的可靠性，但是一个服务端当中假如存在大量的TIME_WAIT状态，那么服务端的工作能力会极大地受到限制，而取消TIME_WAIT状态其实对可靠性的影响比较小，所以用户可以选择使用`setsockopt`函数修改监听套接字的属性，使其可以在TIME_WAIT状态下依然可以`bind`重复的地址（需要在`bind`之前执行）。

```
int setsockopt(int sockfd, int level, int optname,
                      const void *optval, socklen_t optlen);
```

下面是相关的例子：

```
    //...
    int reuse = 1;
    ret =
setsockopt(sfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));
    ERROR_CHECK(ret,-1,"setsockopt");
    //...后续再执行bind等操作
```

select监听socket支持断开重连

服务端除了接收缓冲区和标准输入以外，还有一个操作也会造成阻塞，那就是`accept`操作，实际上服务端可以使用`select`管理监听套接字，检查其全连接队列是否存在已经建好的连接，如果存在连接，那么其**读事件**即`accept`操作便就绪。将监听套接字加入监听会导致服务端的代码发生一个结构变化：

- 每次重新调用`select`之前需要提前准备好要监听的文件描述符，这些文件描述符当中可能会包括新的已连接套接字的文件描述符。
- `select`的第一个参数应当足够大，从而避免无法监听到新的已连接套接字的文件描述符（它们的数值可能会比较大）。
- 需要处理`accept`就绪的情况。

采用上面的调整之后，服务端就可以支持客户端断开重连了。

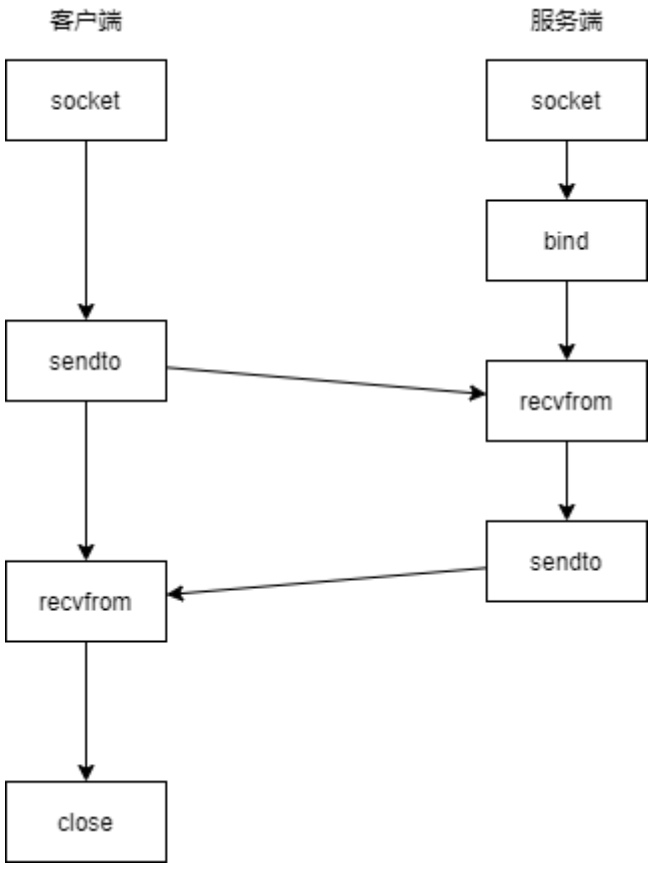
```
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    int newFd=0;
    char buf[64]={0};
    fd_set rdset;
    //记录的是需要监听的文件描述符的集合
    fd_set needMonitorSet;
    FD_ZERO(&rdset);
    FD_ZERO(&needMonitorSet);
    FD_SET(STDIN_FILENO,&needMonitorSet);
    FD_SET(sfd,&needMonitorSet);
    while(1)
    {
        memcpy(&rdset,&needMonitorSet,sizeof(fd_set));
        ret = select(10,&rdset,NULL,NULL,NULL);
        ERROR_CHECK(ret,-1,"readyNum");
        //select的rdset是一个传入传出参数，select成功返回的时候
        //rdset里保存的是就绪的描述符
```

```
        printf("select return\n");
        if(FD_ISSET(STDIN_FILENO,&rdset))
        {
            memset(buf,0,sizeof(buf));
            read(STDIN_FILENO,buf,sizeof(buf));
            send(newFd,buf,strlen(buf)-1,0);
        }
        else if(FD_ISSET(newFd,&rdset))
        {
            memset(buf,0,sizeof(buf));
            ret = recv(newFd,buf,sizeof(buf),0);
            //对端断开的时候，newFd一直可读
            //recv读数据的返回值是0
            if(0 == ret)
            {
                printf("byebye\n");
                close(newFd);
                FD_CLR(newFd,&needMonitorSet);
                continue;
            }
            printf("buf=%s\n",buf);
        }
        else if(FD_ISSET(sfd,&rdset))
        {
            //sfd可读，表示客户端连接我们
            //newFd代表的是跟客户端的TCP连接
            printf("accpet\n");
            struct sockaddr_in cliAddr;
            memset(&cliAddr,0,sizeof(cliAddr));
            socklen_t sockLen = sizeof(cliAddr);
            printf("sockLen=%d\n",sockLen);
            /* socklen_t sockLen = 0; */
            newFd = accept(sfd,(struct
sockaddr*)&cliAddr,&sockLen);
            ERROR_CHECK(newFd,-1,"accept");
            FD_SET(newFd,&needMonitorSet);
            printf("sockLen=%d\n",sockLen);
            printf("newFd=%d\n",newFd);
```

```
 printf("addr=%s,port=%d\n",inet_ntoa(cliAddr.sin_addr),ntohs(cliAd
dr.sin_port));
        }
    }
    close(newFd);
    close(sfd);
    return 0;
}
```

10.4 UDP通信

鸟瞰UDP通信



sendto和recvfrom

这两个函数的行为类似于`send`和`recv`，不过会有一些额外的参数，用来指定或者保存对端的套接字地址结构以及对应的大小。

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                      const struct sockaddr *dest_addr, socklen_t
addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                        struct sockaddr *src_addr, socklen_t
*addrlen);
```

在使用UDP进行的通信的时候，要特别注意的是这是一个无连接的协议。一方面调用`socket`函数的时候需要设置`SOCK_DGRAM`选项，而且因为没有建立连接的过程，所以必须总是由客户端先调用`sendto`发送消息给服务端，这样服务端才能知道对端的地址信息，从进入后续的通信。下面是使用UDP通信的一个例子：

```
//client.c
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_DGRAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    char buf[64]={0};
    socklen_t len = sizeof(serAddr);
    sendto(sfd,"hello udp",9,0,(struct sockaddr*)&serAddr,len);
    sendto(sfd,"hello server",12,0,(struct sockaddr*)&serAddr,len);
    recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&serAddr,&len);
    printf("buf=%s\n",buf);
    close(sfd);
    return 0;
}
```

```c
//server.c
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_DGRAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    char buf[4]={0};
    struct sockaddr_in cliAddr;
    memset(&cliAddr,0,sizeof(cliAddr));
    socklen_t len = sizeof(cliAddr);
    recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&cliAddr,&len);
    printf("buf=%s\n",buf);
    recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&cliAddr,&len);
    printf("buf=%s\n",buf);
    sendto(sfd,"hello client",12,0,(struct sockaddr*)&cliAddr,len);
    close(sfd);
    return 0;
}
```

服务端的运行结果：

```
$ ./server_udp 192.168.135.132 2778
buf=hell
buf=hell
```

可以发现UDP是一种保留消息边界的协议，无论用户态空间分配的空间是否足够`recvfrom`总是会取出一个完整UDP报文，那么没有拷贝的用户态内存的数据会直接丢弃。

> *主机A向主机B发送数据，以TCP方式发送3个包，对方可能会收到几个包？答：任意个*

> *以UDP方式发送3个包，对方可能会收到几个包？答：至多3个*

使用UDP的即时聊天

类似基于TCP的即时聊天通信，使用UDP也可以实现即时聊天通信，考虑到UDP是无连接协议，客户端需要首先发送一个消息让服务端知道客户端的地址信息，然后再使用`select`监听网络读缓冲区和标准输入即可。

```
//client.c
int main(int argc,char* argv[])
{
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_DGRAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    char buf[64]={0};
    socklen_t len = sizeof(serAddr);
    int ret = sendto(sfd,"hello server",12,0,(struct
sockaddr*)&serAddr,len);
    printf("ret=%d\n",ret);
    fd_set rdset;
    while(1){
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(sfd,&rdset);
        select(sfd+1,&rdset,NULL,NULL,NULL);
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            memset(buf,0,sizeof(buf));
            read(STDIN_FILENO,buf,sizeof(buf));
            sendto(sfd,buf,strlen(buf)-1,0,(struct
sockaddr*)&serAddr,len);
        }
        else if(FD_ISSET(sfd,&rdset)){
            memset(buf,0,sizeof(buf));
```

```
            recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&serAddr,&len);
            printf("buf=%s\n",buf);
        }
    }
    close(sfd);
    return 0;
}
```

```
//server.c
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    int sfd = socket(AF_INET,SOCK_DGRAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    char buf[64]={0};
    struct sockaddr_in cliAddr;
    memset(&cliAddr,0,sizeof(cliAddr));
    socklen_t len = sizeof(cliAddr);
    //接收消息，这样才能知道客户端的地址信息
    recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&cliAddr,&len);
    printf("buf=%s\n",buf);
    fd_set rdset;
    while(1){
        FD_ZERO(&rdset);
        FD_SET(STDIN_FILENO,&rdset);
        FD_SET(sfd,&rdset);
        select(10,&rdset,NULL,NULL,NULL);
        if(FD_ISSET(STDIN_FILENO,&rdset)){
            memset(buf,0,sizeof(buf));
            ret = read(STDIN_FILENO,buf,sizeof(buf));
            if(ret == 0){
```

```
                break;
            }
            sendto(sfd,buf,strlen(buf)-1,0,(struct
sockaddr*)&cliAddr,len);
        }
        else if(FD_ISSET(sfd,&rdset))
        {
            memset(buf,0,sizeof(buf));
            ret = recvfrom(sfd,buf,sizeof(buf),0,(struct
sockaddr*)&cliAddr,&len);
            //对端断开的时候，newFd一直可读
            //recv读数据的返回值是0
            if(0 == ret){
                printf("byebye\n");
                close(sfd);
                return 0;
            }
            printf("buf=%s\n",buf);
        }
    }
    close(sfd);
    return 0;
}
```

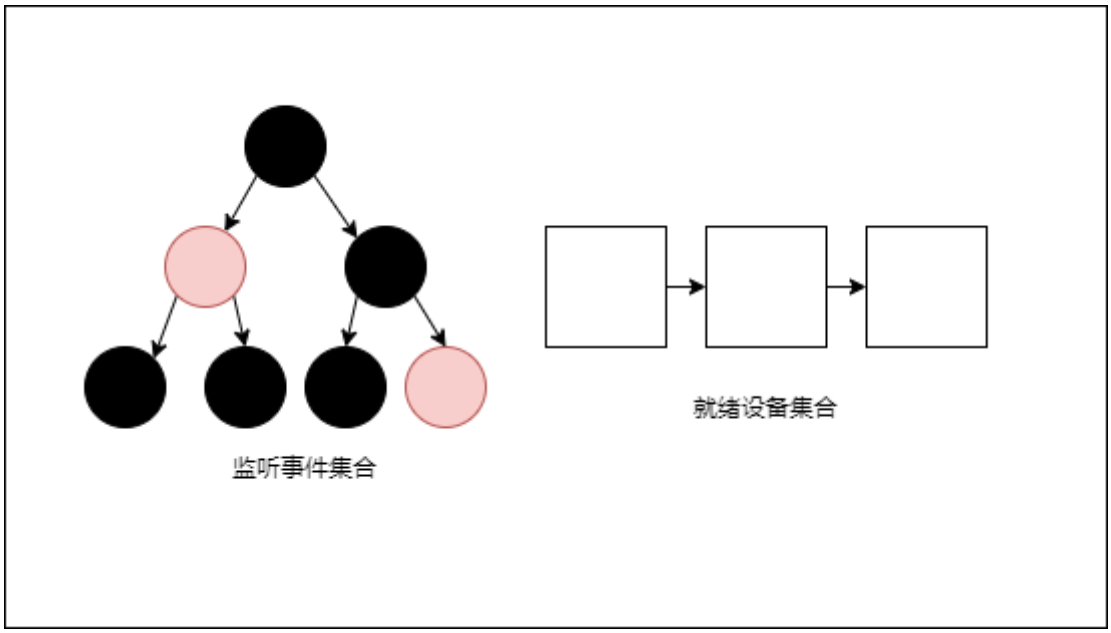
需要特别注意的是，UDP通信不存在连接建立和断开过程，所以服务端无法知道客户端是否已经关闭套接字。

10.5 epoll系统调用

epoll的基本原理

和`select`一样，`epoll`也是一种IO多路复用机制，它可以监听多个设备的就绪状态，让进程或者线程只在有事件发生之后再执行真正的读写操作。`epoll`可以在内核态空间当中维持两个数据结构：**监听事件集合**和**就绪事件队列**。监听事件集合用来存储所有需要关注的设备（即文件描述符）和对应操作（比如读、写、挂起和异常等等），当监听的设备有事件产生时，比如网卡上接收到了数据并传输到了缓冲区当中时，硬件会采用中断等方式通知操作

系统，操作系统会将就绪事件拷贝到就绪事件队列中，并且找到阻塞在`epoll_wait`的线程，让其就绪。监听事件集合通常是一个红黑树，就绪事件队列是一个线性表。



和`select`相比，`epoll`的优势如下：

- 除了水平触发，还支持边缘触发。
- 监听事件集合容量很大，有多少内存就能放下多少文件描述符。
- 监听事件集合常驻内核态，调用`epoll_wait`函数不会修改监听性质，不需要每次将集合从用户态拷贝到内核态。
- 监听事件和就绪事件的状态分为两个数据结构存储，当`epoll_wait`就绪之后，用户可以直接遍历就绪事件队列，而不需要在所有事件当中进行轮询。

有了这些优势之后，`epoll`逐渐取代了`select`的市场地位，尤其是在管理巨大量连接的高并发场景中，`epoll`的性能要远超`select`。

使用epoll取代select

使用`epoll`主要有这样几个函数：

- `epoll_create`用于在内核之中创建一个`epoll`文件对象，这个文件对象中就包含之前所描述的监听事件集合和就绪设备集合。`epoll_create`的参数目前已经没有意义，填写一个大于0的数值即可。`epoll_create`的返回值是该文件对象对应的文件描述符。

- `epoll_ctl`用于调整监听事件集合。`op`的选项是`EPOLL_CTL_ADD`、`EPOLL_CTL_MOD`和`EPOLL_CTL_DEL`，分别表示添加、修改和删除事件，`event->events`用于描述事件的类型，其中`EPOLLIN`表示读，`EPOLLOUT`表示写。可以通过命令`man 7 socket`查看每个操作对应的事件类型如何。
- `epoll_wait`用于使线程陷入阻塞，直到监听的设备就绪或者超时。`events`是一个传入传出参数，用于存储就绪设备队列，`epoll_wait`的返回值就是就绪设备队列的长度，即就绪设备的个数。`timeout`描述超时时间，单位是毫秒，-1是永久等待。`maxevent`传入一个足够大的正数即可。

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout);
typedef union epoll_data {
    void    *ptr;
    int      fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
struct epoll_event {
    uint32_t     events;    /* Epoll events */
    epoll_data_t data;      /* User data variable */
};
```

下面是使用`epoll`取代`select`实现即时聊天通信的例子：

```
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    //创建监听套接字
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
```

```
    int ret =0;
    //reuse=1表示允许地址重用
    int reuse = 1;
    ret =
setsockopt(sfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));
    ERROR_CHECK(ret,-1,"setsockopt");
    ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    //newFd代表的是跟客户端的TCP连接
    int newFd = accept(sfd,NULL,NULL);
    ERROR_CHECK(newFd,-1,"accept");
    char buf[64]={0};
    int epfd = epoll_create(1);
    ERROR_CHECK(epfd,-1,"epoll_create");
    struct epoll_event event,evs[2];
    memset(&event,0,sizeof(event));
    //把关心的描述符和对应的时间填到结构体里
    event.data.fd = STDIN_FILENO;
    event.events = EPOLLIN;
    ret = epoll_ctl(epfd,EPOLL_CTL_ADD,STDIN_FILENO,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl");
    event.data.fd = newFd;
    ret = epoll_ctl(epfd,EPOLL_CTL_ADD,newFd,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl");
    int readyNum=0;
    while(1){
        readyNum = epoll_wait(epfd,evs,2,-1);//不用每次都重置监听集合了
        for(int i=0;i<readyNum;i++){
            if(evs[i].data.fd == STDIN_FILENO){
                memset(buf,0,sizeof(buf));
                read(STDIN_FILENO,buf,sizeof(buf));
                send(newFd,buf,strlen(buf)-1,0);
            }
            else if(evs[i].data.fd == newFd){
                memset(buf,0,sizeof(buf));
                ret = recv(newFd,buf,sizeof(buf),0);
                //对端断开的时候，newFd一直可读
```

```
                //recv读数据的返回值是0
                if(0 == ret){
                    printf("byebye\n");
                    close(sfd);
                    close(newFd);
                    return 0;
                }
                printf("buf=%s\n",buf);
            }
        }
    }
    close(newFd);
    close(sfd);
    return 0;
}
```

使用epoll关闭长期不发消息的连接

和`select`一样，`epoll`也可以监听已连接队列，判断`accept`是否就绪。配合上超时机制，可以用来实现自动断开功能：超过一段时间未发送消息的客户端的TCP连接会被服务端主动关闭。

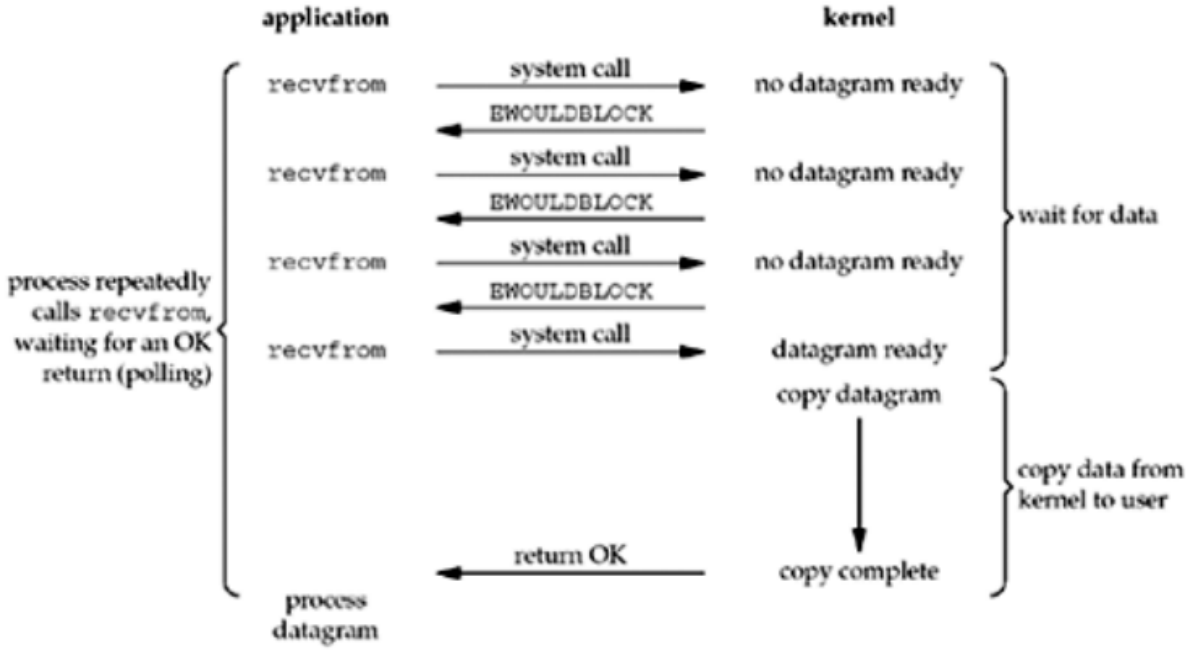
```
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    //创建监听套接字
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret =0;
    //reuse=1表示允许地址重用
    int reuse = 1;
    ret =
setsockopt(sfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));
    ERROR_CHECK(ret,-1,"setsockopt");
```

```
    ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    //newFd代表的是跟客户端的TCP连接
    int newFd = 0;
    char buf[4]={0};
    int epfd = epoll_create(1);
    ERROR_CHECK(epfd,-1,"epoll_create");
    struct epoll_event event,evs[2];
    memset(&event,0,sizeof(event));
    //把关心的描述符和对应的时间填到结构体里
    event.data.fd = STDIN_FILENO;
    event.events = EPOLLIN;
    ret = epoll_ctl(epfd,EPOLL_CTL_ADD,STDIN_FILENO,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl");
    event.data.fd = sfd;
    ret = epoll_ctl(epfd,EPOLL_CTL_ADD,sfd,&event);
    ERROR_CHECK(ret,-1,"epoll_ctl");
    int readyNum=0;
    time_t lastTime,nowTime;
    lastTime = nowTime = time(NULL);
    int isCliLogin = 0;
    while(1){
        readyNum = epoll_wait(epfd,evs,2,1000);
        printf("readyNum=%d \n",readyNum);
        if(0 == readyNum && 1 == isCliLogin ){
            nowTime = time(NULL);
            if(nowTime - lastTime > 5){
                printf("close\n");
                close(newFd);
                isCliLogin = 0;
            }
        }
        for(int i=0;i<readyNum;i++){
            lastTime = time(NULL);
            printf("readyNum=%d fd=%d\n",readyNum,evs[i].data.fd);
            if(evs[i].data.fd == STDIN_FILENO){
                memset(buf,0,sizeof(buf));
```

```
                read(STDIN_FILENO,buf,sizeof(buf));
                send(newFd,buf,strlen(buf)-1,0);
            }
            else if(evs[i].data.fd == newFd){
                    memset(buf,0,sizeof(buf));
                    ret = recv(newFd,buf,sizeof(buf),0);
                    //对端断开的时候，newFd一直可读
                    //recv读数据的返回值是0
                    if(0 == ret){
                        printf("byebye\n");
                        close(newFd);
                        isCliLogin = 0;
                        continue;
                    }
                    printf("buf=%s\n",buf);
            }
            else if(evs[i].data.fd == sfd){
                newFd = accept(sfd,NULL,NULL);
                ERROR_CHECK(newFd,-1,"accept");
                printf("newFd=%d\n",newFd);
                isCliLogin = 1;//表示有客户端登录
                event.data.fd = newFd;
                event.events = EPOLLIN;
                ret = epoll_ctl(epfd,EPOLL_CTL_ADD,newFd,&event);
                ERROR_CHECK(ret,-1,"epoll_ctl");
            }
        }
    }
    close(newFd);
    close(sfd);
    return 0;
}
```

非阻塞读操作

之前所了解的读(`read`)操作都是阻塞的，即调用该函数时，如果硬件没有将数据拷贝到内核缓冲区当中时，线程会主动陷入阻塞。为此，操作系统为用户提供非阻塞版本的`read`：当读取内核缓冲区数据时，如果没有数据，`read`会直接返回-1，并将`errno`的数值设置为`EAGAIN`。非阻塞操作通常会配合循环一起使用以实现一种**同步非阻塞**的IO模型。(下图选自《UNIX网络编程》卷1第6章第2节)。



使用`fcntl`函数可以将已打开文件增加一个非阻塞选项。

```
int fcntl(int fd, int cmd, ... /* arg */ );
//cmd F_GETFL 获取状态作为返回值
//cmd F_SETFL arg STATUS 将文件状态设置为新状态
```

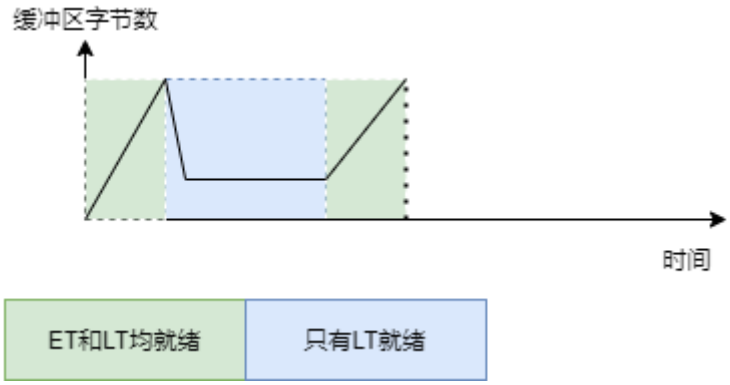
下面是示例：

```
int setNonblock(int fd){
    int ret = 0,status=0;
    //获取描述符的当前的状态
    status = fcntl(fd,F_GETFL);
    status |= O_NONBLOCK;
    ret = fcntl(fd,F_SETFL,status);
    ERROR_CHECK(ret,-1,"fcntl");
    return 0;
}
```

```c
int main(){
    char buf[64]={0};
    setNonblock(STDIN_FILENO);
    //设置为非阻塞的方式后
    //如果有数据，会读数据出来
    //如果没有数据，不会阻塞，会返回-1
    int ret = read(STDIN_FILENO,buf,sizeof(buf));
    printf("ret=%d,buf=%s\n",ret,buf);
    ERROR_CHECK(ret,-1,"read");
    return 0;
}
```

epoll的边缘触发

`epoll_wait`的就绪触发有两种方式：一种是默认的水平触发方式(Level-triggered)，另一种是边缘触发模式(Edge-triggered)。以读事件为例子：水平触发模式下，只要缓冲区当中存在数据，就可以使`epoll_wait`就绪；在边缘触发的情况下，如果缓冲区中存在数据，但是数据一直没有增多，那么`epoll_wait`就不会就绪，只有缓冲区的数据增多的时候，即下图中绿色的上升沿部分时，才能使`epoll_wait`就绪。



使用水平触发的话，线程能够以更短的响应时间来处理事件，但是这可能会导致饥饿问题，如果存在某个事件传输的数据量过大，那么线的epoll_wait就会多次就绪直到处理完所有数据为止，而一些其他的任务所占用的资源就会相对变少。使用边缘触发可以避免这个问题。为了确保读操作可以将所有数据读完，可以考虑使用循环配合非阻塞的形式来处理。

在线程池架构中，主线程通常会将实际的IO交给子线程即工作线程完成，采用边缘触发可以有效地降低主线程的响应频率，提高整体的性能。除此以外，如果一次请求对应一次响应是用户追求的通信模式，那么边缘触发正好符合。

下面是使用边缘触发配合非阻塞IO的例子：

```
            //...
            if(evs[i].data.fd == newFd){
                while(1){//循环配合非阻塞IO
                    memset(buf,0,sizeof(buf));
                    ret = recv(newFd,buf,sizeof(buf),0);
                    //对端断开的时候，newFd一直可读
                    //recv读数据的返回值是0
                    if(0 == ret){
                        printf("byebye\n");
                        close(newFd);
                        break;
                    }
                    else if(-1 == ret){
                        break;
                    }
                    printf("buf=%s\n",buf);
                }
            }
            else if(evs[i].data.fd == sfd){
                newFd = accept(sfd,NULL,NULL);
                ERROR_CHECK(newFd,-1,"accept");
                //把newFd设置为非阻塞的属性
                //搭配边沿触发一起使用
                setNonblock(newFd);
                event.data.fd = newFd;
                //设置为边缘触发的方式
                event.events = EPOLLIN|EPOLLET;
                ret = epoll_ctl(epfd,EPOLL_CTL_ADD,newFd,&event);
                ERROR_CHECK(ret,-1,"epoll_ctl");
            }
            //....
```

10.6 socket属性调整

使用函数`setsocketopt`可以调整套接字的属性，不过需要注意的是具体的属性内容需要参考`man 7 socket`帮助手册才能得知。

```
int setsockopt(int sockfd, int level, int optname,
                       const void *optval, socklen_t optlen);
int getsockopt(int sockfd, int level, int optname,
                       void *optval, socklen_t *optlen);
```

- SO_RCVBUF和SO_SNDBUF：用来获取和调整接收/发送缓冲区的大小。注意到`setsockopt`之后再`getsockopt`的结果会和之前传入的参数不一致。

```
    //...
    int bufsize;
    socklen_t buflen=sizeof(int);
    ret = getsockopt(sfd,SOL_SOCKET,SO_RCVBUF,&bufsize,&buflen);
    ERROR_CHECK(ret,-1,"getsockopt");
    printf("bufsize=%d\n",bufsize);
    bufsize = 8192;
    ret =
setsockopt(sfd,SOL_SOCKET,SO_RCVBUF,&bufsize,sizeof(int));
    ERROR_CHECK(ret,-1,"setsockopt");
    ret = getsockopt(sfd,SOL_SOCKET,SO_RCVBUF,&bufsize,&buflen);
    ERROR_CHECK(ret,-1,"getsockopt");
    printf("bufsize2=%d\n",bufsize);
    //...
```

- SO_RCVLOWAT和SO_SNDLOWAT：这个参数说明一个缓冲区的下限，如果缓冲区的字节少于下限，那么数据就不会从套接字中传递给内核协议栈或者发送给用户。

```
            //...
            else if(evs[i].data.fd == sfd)
            {
                newFd = accept(sfd,NULL,NULL);
                ERROR_CHECK(newFd,-1,"accept");
                printf("newFd=%d\n",newFd);
                int buflowat= 10;
```

```
                //设置接收缓冲区下限
                ret =
setsockopt(newFd,SOL_SOCKET,SO_RCVLOWAT,&buflowat,sizeof(int));
                ERROR_CHECK(ret,-1,"setsockopt");
                flag = 1;//表示有客户端登录
                event.data.fd = newFd;
                event.events = EPOLLIN;
                ret = epoll_ctl(epfd,EPOLL_CTL_ADD,newFd,&event);
                ERROR_CHECK(ret,-1,"epoll_ctl");
            }
            //...
            //这样修改了之后，发送方的数据量如果比较少，将不会触发epoll_wait
的读就绪
```

10.7 recv和send的标志

MSG_DONTWAIT

`recv`的本标志可以在不修改已连接套接字的文件属性的情况下，把单个IO操作临时指定为非阻塞，随后执行IO操作，最后关闭非阻塞标志。

```
            //...
            else if(evs[i].data.fd == newFd){
                while(1){
                    memset(buf,0,sizeof(buf));
                    //MSG_DONTWAIT表示把recv改为非阻塞的方式
                    ret = recv(newFd,buf,sizeof(buf),MSG_DONTWAIT);
                    //对端断开的时候，newFd一直可读
                    //recv读数据的返回值是0
                    if(0 == ret){
                        printf("byebye\n");
                        close(newFd);
                        continue;
                    }
                    else if(-1 == ret){
```

```
                    break;
                }
                printf("buf=%s\n",buf);
            }
        }
        else if(evs[i].data.fd == sfd){
            newFd = accept(sfd,NULL,NULL);
            ERROR_CHECK(newFd,-1,"accept");
            //把newFd设置为非阻塞的属性
            //搭配边沿触发一起使用
            event.data.fd = newFd;
            //设置为边沿触发的方式
            event.events = EPOLLIN|EPOLLET;
            ret = epoll_ctl(epfd,EPOLL_CTL_ADD,newFd,&event);
            ERROR_CHECK(ret,-1,"epoll_ctl");

        }
        //...
```

MSG_PEEK

`recv`的MSG_PEEK选项可以允许看到已经到达缓冲区的数据而不将其从缓冲区当中取出。如果配合MSG_DONTWAIT一起使用，可以得知当前正在排队的数量到底有多少。

```
int main(int argc,char* argv[]){
    ARGS_CHECK(argc,3);
    //创建监听套接字
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    ERROR_CHECK(sfd,-1,"socket");
    struct sockaddr_in serAddr;
    memset(&serAddr,0,sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
    int ret = bind(sfd,(struct sockaddr*)&serAddr,sizeof(serAddr));
    ERROR_CHECK(ret,-1,"bind");
```

```
    ret = listen(sfd,10);
    ERROR_CHECK(ret,-1,"listen");
    //newFd代表的是跟客户端的TCP连接
    int newFd = accept(sfd,NULL,NULL);
    ERROR_CHECK(newFd,-1,"accept");
    ret = send(newFd,"helloclient",11,0);
    ERROR_CHECK(ret,-1,"send");
    char buf[64]={0};
    //recv添加了MSG_PEEK之后，到缓冲区中读数据的时候
    //只是把缓冲区中的数据做一份拷贝
    //并不会真正的移走数据，这样下一次recv的时候
    //还可以到缓冲区里拿到这些数据
    ret = recv(newFd,buf,sizeof(buf),MSG_PEEK);
    /* ret = recv(newFd,buf,sizeof(buf),0); */
    printf("recv1 ret=%d,buf=%s\n",ret,buf);
    memset(buf,0,sizeof(buf));
    ret = recv(newFd,buf,sizeof(buf),0);
    printf("recv2 ret=%d,buf=%s\n",ret,buf);
    close(newFd);
    close(sfd);
    return 0;
}
```